8 Common protected resources vulnerabilities

This chapter covers

- Avoiding common implementation vulnerabilities in protected resources
- Counting known attacks against protected resources
- Benefiting from modern browser protections when designing a protected resource's endpoint

In the previous chapter, we reviewed common attacks against OAuth clients. Now it's time to see how to protect a resource server and defend against common attacks targeting OAuth protected resources. In this chapter, we're going to learn how to design resource endpoints to minimize the risk of token spoofing and token replay. We'll also see how we can leverage modern browsers' protection mechanisms to make the designer's life easier.

8.1 How are protected resources vulnerable?

Protected resources are vulnerable in a number of ways, the first and most obvious being that access tokens can leak, giving an attacker data about protected resources. This can happen via token hijacking, as seen in the previous chapter, or because the token has weak entropy or an overly wide scope. Another issue related to protected resources is that the endpoints can be vulnerable to cross-site scripting (XSS) attacks. Indeed, if the resource server chooses to support `access_token` as a URI

[redacted]

parameter,[1] the attacker can forge a URI containing the XSS attack and then use social engineering to trick a victim into following that link.[2] This can be as simple as a blog post with a review of the application, inviting people to try it out. When someone clicks on that link, the malicious JavaScript is then executed.

What is XSS?

Cross-site scripting (XSS) is the Open Web Application Security Project's (OWASP) Top Ten number three[3] and is by far the most prevalent web application security flaw. Malicious scripts are injected into otherwise benign and trusted websites to bypass access controls such as the same-origin policy. As a result, an attacker might inject a script and modify the web application to suit their own purposes, such as extracting data that will allow the attacker to impersonate an authenticated user or perhaps to input malicious code for the browser to execute.

8.2 Design of a protected resource endpoint

Designing a web API is a rather complex exercise (as it is for any API) and many factors should be taken in consideration. In this section, you'll learn how to design a safe web API that leverages all the help provided in a modern browser. If you're designing a REST API in which the result response is driven by some user input, the risk of encountering an XSS vulnerability is high. We need to leverage as much as possible the features provided by modern browsers in combination with some common best practices at any point that a resource is exposed on the web.

As a concrete example, we're going to introduce a new endpoint (`/helloWorld`) together with a new scope (`greeting`). This new API will look like:

```
GET /helloWorld?language={language}
```

This endpoint is rather simple: it greets the user in language taken as input. Currently supported languages are found in table 8.1. Other language inputs will produce an error.

Table 8.1 Supported languages in our test API

Key	Value
en	English
de	German
it	Italian
fr	French
es	Spanish

[1] RFC 6750 https://tools.ietf.org/html/rfc6750#section-2.3
[2] http://intothesymmetry.blogspot.ch/2014/09/bounty-leftover-part-2-target-google.html
[3] https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS)

[redacted]

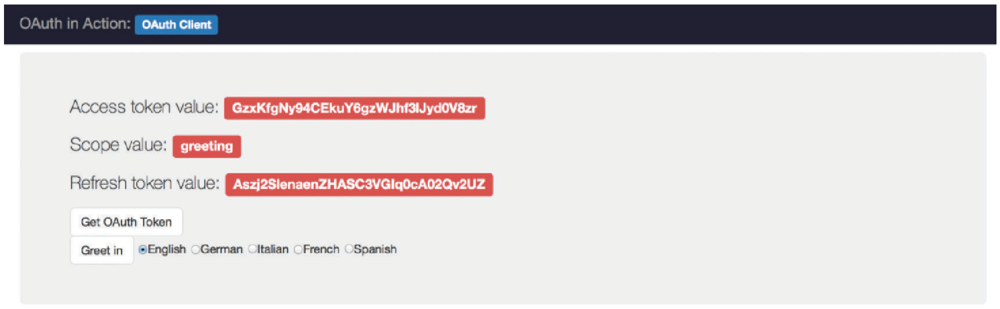
8.2.1 How to protect a resource endpoint

You can see an implementation of this endpoint in `ch-8-ex-1`. Open up the `protectedResource.js` file in that folder. If you scroll down to the bottom of this file, you can see the fairly simple implementation of our functionality.

```
app.get("/helloWorld", getAccessToken, function(req, res){
   if (req.access_token) {
         if (req.query.language == "en") {
                   res.send('Hello World');
         } else if (req.query.language == "de") {
                   res.send('Hallo Welt');
         } else if (req.query.language == "it") {
                   res.send('Ciao Mondo');
         } else if (req.query.language == "fr") {
                   res.send('Bonjour monde');
         } else if (req.query.language == "es") {
                   res.send('Hola mundo');
         } else {
                   res.send("Error, invalid language: "+ req.query.language);
         }
   }
});
```

To give the previous example a try, run all three components simultaneously and do the usual "OAuth dance" as in figure 8.1.

By clicking on the Greet In button, you can then request a greeting in English, which causes the client to call the protected resource and display the results (figure 8.2).

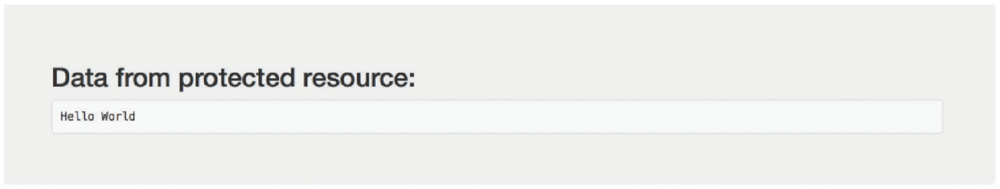

OAuth in Action: OAuth Client

Access token value: GzxKfgNy94CEkuY6gzWJhf3lJyd0V8zr

Scope value: greeting

Refresh token value: Aszj2SlenaenZHASC3VGlq0cA02Qv2UZ

Figure 8.1 Access token with greeting scope


Data from protected resource:

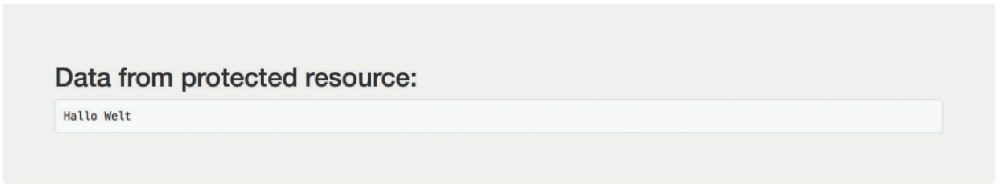
```
Hello World
```

Figure 8.2 Greeting in English

[redacted]

Selecting a different language (for example, German) will then display what we see in figure 8.3.

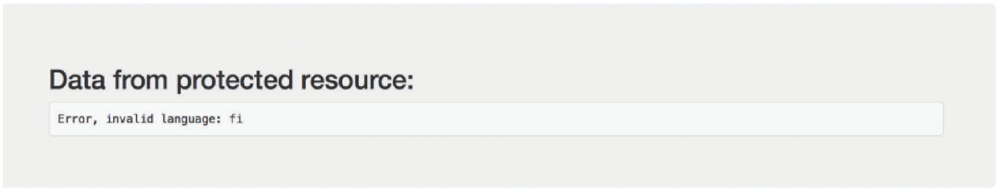
If the language isn't supported, an error message will be shown, as in figure 8.4.



Data from protected resource:

```
Hallo Welt
```

Figure 8.3 Greeting in German



Data from protected resource:

```
Error, invalid language: fi
```

Figure 8.4 Invalid language

It's also possible to directly hit the resource endpoint passing an `access_token` by using a command line HTTP client such as curl:[4]

```
> curl -v -H "Authorization: Bearer TOKEN"
http://localhost:9002/helloWorld?language=en
```

Or leveraging the previous URI parameter support for `access_token`:

```
> curl -v "http://localhost:9002/helloWorld?access_token=TOKEN&language=en"
```

In both cases, the result will be something like the following response that shows a greeting in English:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 11
Date: Mon, 25 Jan 2016 21:23:26 GMT
Connection: keep-alive

Hello World
```

Now let's try hitting the `/helloWorld` endpoint by passing an invalid language:

```
> curl -v "http://localhost:9002/helloWorld?access_token=TOKEN&language=fi"
```

[4] https://curl.haxx.se/

[redacted]

The response is something like the following, which shows an error message because Finnish isn't one of the supported languages:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 27
Date: Tue, 26 Jan 2016 16:25:00 GMT
Connection: keep-alive

Error, invalid language: fi
```

So far, so good. But as any bug hunter will notice, it seems that the error response of the `/helloWorld` endpoint is designed in a way that the erroneous input bounces back into the response. Let's try to push this further and pass a nasty payload.

```
> curl -v  "http://localhost:9002/helloWorld?access_token=TOKEN&language=<sc
ript>alert('XSS')</script>"
```

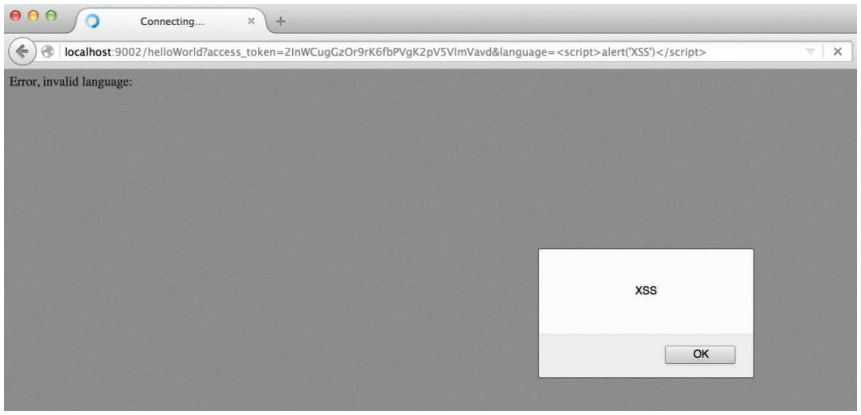
which will yield:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 59
Date: Tue, 26 Jan 2016 17:02:16 GMT
Connection: keep-alive

Error, invalid language: <script>alert('XSS')</script>
```

As you can see, the provided payload is returned verbatim and unsanitized. At this point, the suspicion that this endpoint is susceptible to XSS is more than likely true, and the next step is pretty simple. In order to exploit this, an attacker would forge a malicious URI pointing to the protected resource:

```
http://localhost:9002/helloWorld?access_token=TOKEN&language=<script>alert(
'XSS')</script>
```

When the victim clicks on it, the attack is completed, forcing the JavaScript to execute (figure 8.5).


Connecting...

localhost:9002/helloWorld?access_token=2InWCugGzOr9rK6fbPVgK2pV5VImVavd&language=<script>alert('XSS')</script>

Error, invalid language:

XSS

OK

Figure 8.5 XSS in the protected resource endpoint

[redacted]

Of course, a real attack would not contain a simple JavaScript alert, but would rather use some malicious code that, for example, would extract data to allow the attacker to impersonate an authenticated user. Our endpoint is clearly vulnerable to XSS attack, so we need to fix it. At this point, the recommended approach is to properly escape all untrusted data. We're using URL encoding here.

```
app.get("/helloWorld", getAccessToken, function(req, res){
   if (req.access_token) {
         if (req.query.language == "en") {
                res.send('Hello World');
         } else if (req.query.language == "de") {
                res.send('Hallo Welt');
         } else if (req.query.language == "it") {
                res.send('Ciao Mondo');
         } else if (req.query.language == "fr") {
                res.send('Bonjour monde');
         } else if (req.query.language == "es") {
                res.send('Hola mundo');
         } else {
                res.send("Error, invalid language: "+
                   querystring.escape(req.query.language));
         }
   }
});
```

With this fix in place now, the error response of the forged request would be something like the following:

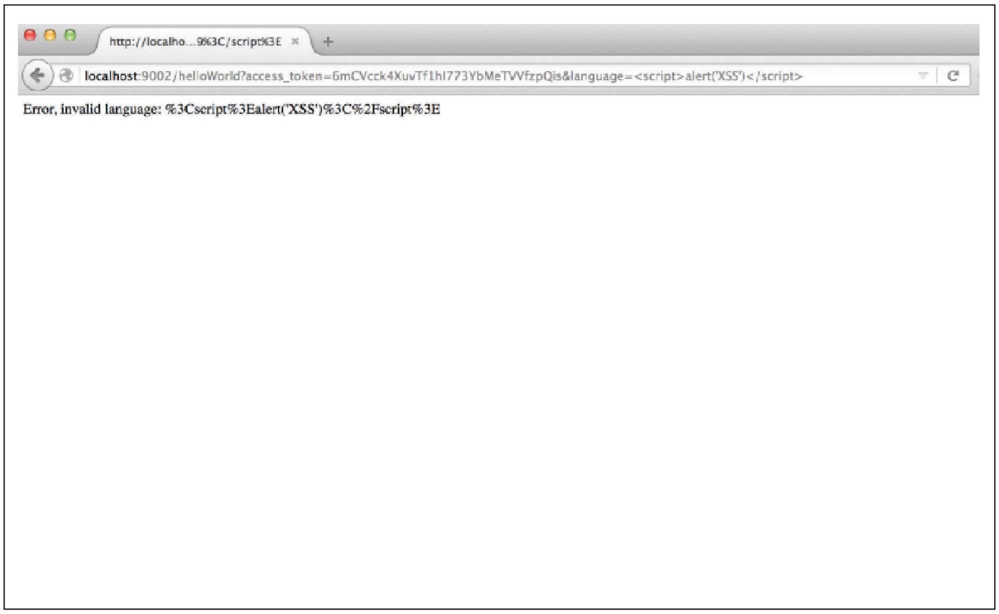
```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 80
Date: Tue, 26 Jan 2016 17:36:29 GMT
Connection: keep-alive

Error, invalid language:
%3Cscript%3Ealert(%E2%80%98XSS%E2%80%99)%3C%2Fscript%3E
```

Consequently, the browser will render the response without executing the rouge script (figure 8.6). Are we done? Well, not quite yet. Output sanitization is the preferred approach for defending against XSS, but is it the only one? The problem with output sanitization is that developers often forget about it, and even if they forget to validate one single input field, we're back to square one in terms of XSS protection. Browser vendors try hard to stop XSS and ship a series of features as mitigation, one of the most important being returning the right `Content-Type` for the protected resource endpoint.

By definition,[5] the `Content-Type` entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

[5] RFC 7231 https://tools.ietf.org/html/rfc7231#section-3.1.1.5

[redacted]



```
http://localho...9%3C/script%3E
localhost:9002/helloWorld?access_token=6mCVcck4XuvTf1hl773YbMeTVVfzpQis&language=<script>alert('XSS')</script>
Error, invalid language: %3Cscript%3Ealert('XSS')%3C%2Fscript%3E
```

Figure 8.6 Sanitized response in the protected resource endpoint

Returning the proper `Content-Type` might save a lot of headaches. Returning to our original unsanitized `/helloWorld` endpoint, let's see how we can improve the situation. The original response looked like this:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 27
Date: Tue, 26 Jan 2016 16:25:00 GMT
Connection: keep-alive

Error, invalid language: fi
```

Here, the `Content-Type` is `text/html`. This might explain why the browser happily executed the rogue JavaScript in the shown XSS attack. Let's try using a different Content-Type like `application/json`:

```
app.get("/helloWorld", getAccessToken, function(req, res){
  if (req.access_token) {

        var resource = {
                "greeting" : ""
        };
        if (req.query.language == "en") {
                resource.greeting = 'Hello World';
        } else if (req.query.language == "de") {
                resource.greeting ='Hallo Welt';
        } else if (req.query.language == "it") {
                resource.greeting = 'Ciao Mondo';
```

[redacted]

```
        } else if (req.query.language == "fr") {
              resource.greeting = 'Bonjour monde';
        } else if (req.query.language == "es") {
              resource.greeting ='Hola mundo';
        } else {
              resource.greeting = "Error, invalid language: "+
              req.query.language;
        }
        res.json(resource);
   }
});
```

In this case,

```
> curl -v "http://localhost:9002/helloWorld?access_token=TOKEN&language=en"
```

will return

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 33
Date: Tue, 26 Jan 2016 20:19:05 GMT
Connection: keep-alive

{"greeting": "Hello World"}
```

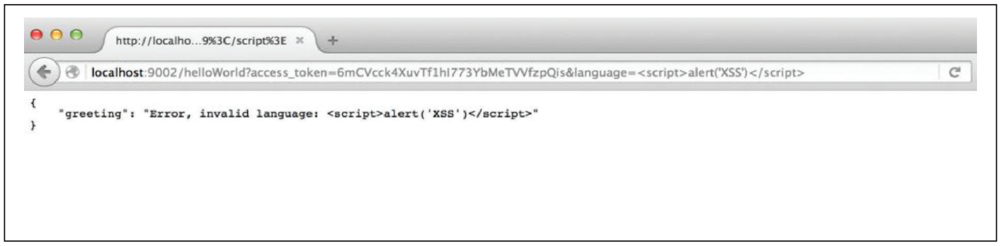
and

```
> curl -v   "http://localhost:9002/helloWorld?access_token=TOKEN&language=<sc
ript>alert('XSS')</script>"
```

will yield the following output:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 76
Date: Tue, 26 Jan 2016 20:21:15 GMT
Connection: keep-alive

{"greeting": "Error, invalid language: <script>alert('XSS')</script>" }
```

Notice that the output string isn't sanitized or encoded in any way, but it is now enclosed in a JSON string value. If we try this straight in the browser, we can appreciate that having the proper `Content-Type` immediately stops the attack on its own (figure 8.7).



```
localhost:9002/helloWorld?access_token=6mCVcck4XuvTf1hl773YbMeTVVfzpQis&language=<script>alert('XSS')</script>
```

```
{
   "greeting": "Error, invalid language: <script>alert('XSS')</script>"
}
```

Figure 8.7 Content-Type application/json in the protected resource endpoint

[redacted]

This happens because browsers respect the "contract" associated with the `application/json Content-Type` and refuse to execute JavaScript if the returned resource is in this form. It's still entirely possible for a poorly written client application to inject the JSON output into an HTML page without escaping the string, which would lead to the execution of the malicious code. As we said, this is just a mitigation, and it's still a good practice to always sanitize the output. We combine these into the following:

```
app.get("/helloWorld", getAccessToken, function(req, res){
   if (req.access_token) {

         var resource = {
                "greeting" : ""
         };
         if (req.query.language == "en") {
                resource.greeting = 'Hello World';
         } else if (req.query.language == "de") {
                resource.greeting ='Hallo Welt';
         } else if (req.query.language == "it") {
                resource.greeting = 'Ciao Mondo';
         } else if (req.query.language == "fr") {
                resource.greeting = 'Bonjour monde';
         } else if (req.query.language == "es") {
                resource.greeting ='Hola mundo';
         } else {
                resource.greeting = "Error, invalid language: "+ querystring.
                escape(req.query.language);
         }
         }
         res.json(resource);
   }
});
```

This is definitely an improvement, but there is still something more we can do to dial the security up to eleven. One other useful response header supported by all the browsers, with the exception of Mozilla Firefox, is X-Content-Type-Options: nosniff. This security header was introduced by Internet Explorer[6] to prevent browsers from MIME-sniffing a response away from the declared Content-Type (just in case). Another security header is X-XSS-Protection, which automatically enables the XSS filter built into most recent web browsers (again with the exception of Mozilla Firefox). Let's see how we can integrate these headers in our endpoint.

```
app.get("/helloWorld", getAccessToken, function(req, res){
   if (req.access_token) {

         res.setHeader('X-Content-Type-Options', 'nosniff');
         res.setHeader('X-XSS-Protection', '1; mode=block');

         var resource = {
                "greeting" : ""
         };
```

[6] https://blogs.msdn.microsoft.com/ie/2008/09/02/ie8-security-part-vi-beta-2-update

[redacted]

```
        if (req.query.language == "en") {
               resource.greeting = 'Hello World';
        } else if (req.query.language == "de") {
               resource.greeting ='Hallo Welt';
        } else if (req.query.language == "it") {
               resource.greeting = 'Ciao Mondo';
        } else if (req.query.language == "fr") {
               resource.greeting = 'Bonjour monde';
        } else if (req.query.language == "es") {
               resource.greeting ='Hola mundo';
        } else {
               resource.greeting = "Error, invalid language: "+ querystring.
               escape(req.query.language);
        }
        res.json(resource);
   }
});
```

Our response will look like this:

```
HTTP/1.1 200 OK
X-Powered-By: Express
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Content-Type: application/json; charset=utf-8
Content-Length: 102
Date: Wed, 27 Jan 2016 17:07:50 GMT
Connection: keep-alive

{
    "greeting": "Error, invalid language:
    %3Cscript%3Ealert(%E2%80%98XSS%E2%80%99)%3C%2Fscript%3E"
}
```

Some room for improvement exists here and it's called the Content Security Policy (CSP).[7] This is yet another response header (Content-Security-Policy) that, quoting the specification, "helps you reduce XSS risks on modern browsers by declaring what dynamic resources are allowed to load via a HTTP Header." This topic deserves a chapter of its own and isn't the main focus of this book; including the proper CSP header field is left as an exercise for the reader.

A resource server can do one final thing to eliminate any chance that a particular endpoint is susceptible to XSS: choose not to support the `access_token` being passed as a request parameter.[8] Doing so would make an XSS on the endpoint theoretically possible but not exploitable because there is no way an attacker can forge a URI that also contains the access token (now expected to be sent in the `Authorization: Bearer` header). This might sound too restrictive, and there might be valid cases in which using this request parameter is the only possible

[7] http://content-security-policy.com/
[8] RFC 6750 https://tools.ietf.org/html/rfc6750#section-2.3

[redacted]

solution in a particular situation. However, all such cases should be treated as exceptions and approached with proper caution.

8.2.2 *Adding implicit grant support*

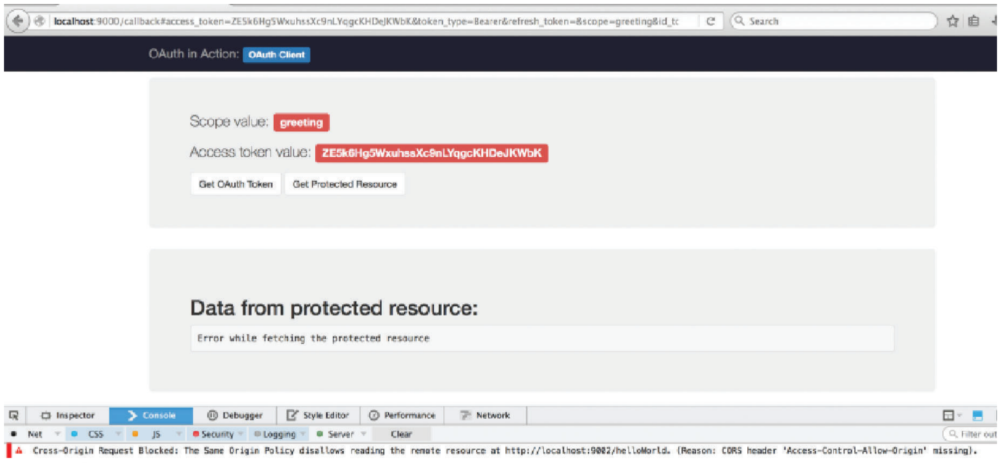
Let's implement a resource endpoint that is also ready to serve an OAuth client supporting the "Implicit Grant" flow described in detail in chapter 6. All the security concerns discussed in the previous section stand, but we need to take care of some extra factors. Open up the `ch-8-ex-2` folder and execute the three Node.js files.

Now open your browser `http://127.0.0.1:9000` and go ahead with the usual "OAuth dance". However, when you try to get the resource, you'll encounter an issue (see figure 8.8).

If you open the browser's JavaScript console (or an equivalent debugging tool), you'll see an error displayed:

> *Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at http://localhost:9002/helloWorld. (Reason: CORS header 'Access-Control-Allow-Origin' missing).*

What, then, is that all about? The browser is trying to tell us that we're attempting to do something illegal: we're trying to use JavaScript to call a URL with a different origin, hence violating the *same origin policy*[9] that browsers enforce. In particular, from the implicit client running on `http://127.0.0.1:9000`, we're trying to implement an AJAX request to `http://127.0.0.1:9002`. In essence, the same origin policy states that "browser windows can work in contexts of each other only if they are from served from the same base URL, consisting of `protocol://domain:port`." We're clearly



```
localhost:9000/callback#access_token=ZE5k6Hg5WxuhssXc9nLYqgcKHDeJKWbK&token_type=Bearer&refresh_token=&scope=greeting&id_tc
```

Scope value: greeting

Access token value: ZE5k6Hg5WxuhssXc9nLYqgcKHDeJKWbK

Data from protected resource:

```
Error while fetching the protected resource
```

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at http://localhost:9002/helloWorld. (Reason: CORS header 'Access-Control-Allow-Origin' missing).
```

Figure 8.8 Same Origin Policy issue

[9] https://en.wikipedia.org/wiki/Same-origin_policy

[redacted]

violating it in this case because the ports don't match: `9000` versus `9002`. On the web, it's more common that a client application be served from one domain, whereas a protected resource be served from another domain, as in our photo-printing example.

Same origin policy in Internet Explorer

The error in exercise 8.1 doesn't show up in Internet Explorer. The reason behind it is described in `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#IE_Exceptions`. In a nutshell, Internet Explorer doesn't include port into Same Origin components; therefore, `http://localhost:9000` and `http://localhost:9002` are considered to be from the same origin and no restrictions are applied. This is different from all other major browsers and, in the opinion of the authors, fairly daft.

The same origin policy is set up to keep JavaScript inside one page from loading malicious content from another domain. But in this case, it's fine to allowing a JavaScript call to our API, especially since we're protecting that API with OAuth to begin with. To solve this, we get a solution straight from the W3C specification:[10] cross-origin resource sharing (CORS). Adding CORS support in Node.js is extremely simple, and it's becoming commonly available in many languages and platforms. Open up and edit the `protectedResource.js` file in the `ch-8-ex-2` folder and require the CORS library:

```
var cors = require('cors');
```

Then add that function as a filter ahead of the other functions. Note that we also add support for the HTTP OPTIONS verb here, which lets our JavaScript client fetch important headers, including the CORS headers, without doing a full request.

```
app.options('/helloWorld', cors());
app.get("/helloWorld", cors(), getAccessToken, function(req, res){
  if (req.access_token) {
```

The rest of the processing code doesn't need to change at all. Now when we try to do the full round-trip, we will achieve the desired result (see figure 8.9).

To understand why everything went through smoothly this time, let's look at the HTTP call that our client made to the protected resource. Using the curl again allows us to see all the headers.

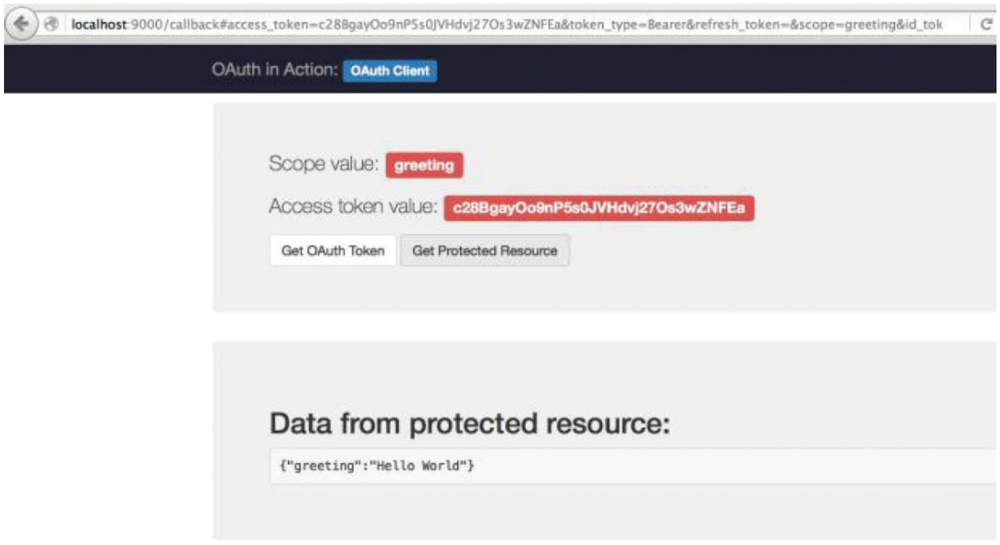
```
> curl -v -H "Authorization: Bearer TOKEN"
http://localhost:9002/helloWorld?language=en
```

now gives

```
HTTP/1.1 200 OK
X-Powered-By: Express
```

[10] https://www.w3.org/TR/cors/

[redacted]



localhost:9000/callback#access_token=c28BgayOo9nP5s0JVHdvj27Os3wZNFEa&token_type=Bearer&refresh_token=&scope=greeting&id_tok

OAuth in Action: OAuth Client

Scope value: greeting

Access token value: c28BgayOo9nP5s0JVHdvj27Os3wZNFEa

Data from protected resource:

```
{"greeting":"Hello World"}
```

Figure 8.9 Protected resource with CORS enabled

```
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Content-Type: application/json; charset=utf-8
Content-Length: 33
Date: Fri, 29 Jan 2016 17:42:01 GMT
Connection: keep-alive

{
    "greeting": "Hello World"
}
```

This new header tells our browser, which is hosting the JavaScript application, that it's OK to allow any origin to call this endpoint. This provides a controlled exception to the same origin policy. It makes sense to apply this on APIs such as protected resources, but to keep it off (the default in most systems) for user-interactive pages and forms.

CORS is a relatively new solution and was not always available in browsers. In the past, the preferred option was to use JSON with Padding[11] (known as JSONP). JSONP is a technique used by web developers to overcome the cross-domain restrictions imposed by browsers to allow data to be retrieved from systems other than the one the page was served by, but it's merely a trick. Effectively, JSON data is presented as a JavaScript script to be loaded and executed in the target environment, usually by use of a specified callback function. Because the data call is presented as a script and not an

[11] https://en.wikipedia.org/wiki/JSONP

[redacted]

AJAX call, the browser bypasses its same origin policy checks. Over the years, usage of JSONP has been abandoned in favor of CORS due to vulnerabilities that used JSONP as a vector (Rosetta Flash in primis[12]). For this reason, we aren't going to include an example with the protected resource endpoint supporting JSONP.

The Rosetta Flash exploit

Rosetta Flash is an exploitation technique discovered and published by Google security engineer Michele Spagnuolo in 2014. It allows an attacker to exploit servers with a vulnerable JSONP endpoint by causing Adobe Flash Player to believe that an attacker-specified Flash applet originated on the vulnerable server. To hinder this attack vector in most modern browsers, it's possible to return the HTTP header X-Content-Type-Options: nosniff and/or prepend the reflected callback with `/**/`.

8.3 Token replays

In the previous chapter, we saw how it's possible to steal an access token. Even if the protected resource runs over HTTPS, once the attacker gets their hands on the access token they will be able to access the protected resource. For this reason, it's important to have an access token that has a relatively short lifetime to minimize the risk of token replay. Indeed, even if an attacker manages to get hold of a victim access token, if it has already expired (or is close to being expired) the severity of the attack decreases. We cover protecting tokens in depth in chapter 10.

One of the main differences of OAuth 2.0 and its predecessor is the fact that the core framework is free of cryptography. Instead, it relies completely on the presence of Transport Layer Security (TLS) across the various connections. For this reason, it's considered best practice to enforce the usage of TLS as much as possible throughout an OAuth ecosystem. Again, another standard comes to the rescue: HTTP Strict Transport Security (HSTS)[13] defined in RFC6797.[14] HSTS allows web servers to declare that browsers (or other complying user agents) should interact with it only using secure HTTPS connections, never via the insecure HTTP protocol. Integrating HSTS in our endpoint is straightforward and, like CORS, requires adding a couple of extra headers. Open up and edit the `protectedResource.js` file in the `ch-8-ex-3` folder to add the appropriate header.

```
app.get("/helloWorld", cors(), getAccessToken, function(req, res){
  if (req.access_token) {

      res.setHeader('X-Content-Type-Options','nosniff');
      res.setHeader('X-XSS-Protection', '1; mode=block');
```

[12] https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/
[13] https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security
[14] RFC 6797 https://tools.ietf.org/html/rfc6797

[redacted]

```
        res.setHeader('Strict-Transport-Security', 'max-age=31536000');

        var resource = {
               "greeting" : ""
        };

        if (req.query.language == "en") {
               resource.greeting = 'Hello World';
        } else if (req.query.language == "de") {
               resource.greeting ='Hallo Welt';
        } else if (req.query.language == "it") {
               resource.greeting = 'Ciao Mondo';
        } else if (req.query.language == "fr") {
               resource.greeting = 'Bonjour monde';
        } else if (req.query.language == "es") {
               resource.greeting ='Hola mundo';
        } else {
               resource.greeting = "Error, invalid language: "+ querystring.
               escape(req.query.language);
        }
        res.json(resource);
   }
});
```

and now when you try to hit the `/helloWorld` endpoint from an HTTP client:

```
> curl -v -H "Authorization: Bearer TOKEN"
http://localhost:9002/helloWorld?language=en
```

you can notice the HSTS response header

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Strict-Transport-Security: max-age=31536000
Content-Type: application/json; charset=utf-8
Content-Length: 33
Date: Fri, 29 Jan 2016 20:13:06 GMT
Connection: keep-alive

{
    "greeting": "Hello World"
}
```

At this point, every time you try to hit the endpoint with the browser using HTTP (not over TLS), you would notice an internal 307 redirect made from the browser. This will avoid any unexpected unencrypted communication (like protocol downgrade attacks). Our test environment doesn't use TLS at all, so this header effectively makes our resource completely inaccessible. Although this is, of course, very secure, it's not particularly useful as a resource. A production system with a real API will need to balance both security and accessibility.

[redacted]

8.4 Summary

We end with some takeaways to ensure a secure protected resource.

- Sanitize all untrusted data in the protected resource response.
- Choose the appropriate Content-Type for the specific endpoint.
- Leverage browser protection and the security headers as much as you can.
- Use CORS if your protected resource's endpoint needs to support the implicit grant flow.
- Avoid having your protected resource support JSONP (if you can).
- Always use TLS in combination with HSTS.

Now that we've secured the client and the protected resource, let's take a look at what it takes to secure the most complex component of the OAuth ecosystem: the authorization server.

[redacted]